HPP Individual Assignment: Parallel quicksort

Olof Björck

March 21, 2019

1 Introduction

Sorting algorithms are prevalent in computer science. A popular and efficient sorting algorithm is quicksort [1], which has an average performance of $O(N \log N)$. The quicksort algorithm of an array can be described as the following:

- Select an element from the array, called the pivot.
- Position the pivot and reorder the array such that all elements less than the pivot are to the left of the pivot, and all elements greater than the pivot are to the right of the pivot. The pivot is now in it's sorted position. This process is called partitioning.
- Recursively perform quicksort on the sub-array to the left of the pivot and on the sub-array to the right of the pivot, recursing as long as such sub-arrays exist.

The C standard library `<stdlib.h>` includes an efficient implementation of the quicksort algorithm through the function `qsort`. However, `qsort` is not parallelized. In this report, a parallel quicksort algorithm for integers is implemented and evaluated.

2 Problem description

To make use of parallelism for the quicksort algorithm, the straightforward, naive implementation would be to let a new thread handle each new sub-array in each partition. This approach raises one main concern: There is limited parallelism as the first quicksort pass is performed by only one processor. Furthermore, the first quicksort pass is the most expensive as it passes through all N array elements. [2] A non-naive parallel version of quicksort is implemented that makes good use of parallelism is implemented in this report.

3 Solution method

A more efficient parallel version of quicksort than the naive version for an array can be stated as follows:

- Divide the array into one part for each thread.
- In each thread, quicksort the corresponding part of the array.
-
  - Select a pivot in each thread, for the corresponding part of the array;
  - Divide the threads into two groups: one group for the elements lower or equal to the pivots, one group for the elements higher than the pivots; and
  - Merge elements pairwise from the threads so that the thread groups are created and in each thread, the corresponding part of the array is sorted.
- Recursively perform the pivot selection, thread grouping, and merging for the groups until the array is sorted.

This algorithm performs better than the naive version because it makes better use of parallelism. It also reduces data dependencies to be pairwise between threads. An important thing to note is that this algorithm only works for number of threads that are a power of two. In the implementation in this report, such a check is performed and the number of threads used is adjusted if necessary. If only one thread is used, the sort is performed using the regular quicksort in `qsort`.

3.1 Optimizations attempted

3.1.1 General code optimizations

Some general code optimizations were used:

- Keyword `__attribute__((pure))` was used for one function, the comparison function used in `qsort`. However, no speedup was observed.
- Keyword `__restrict` was used where applicable. Some speedup was observed by this.
- Keyword `const` was used where applicable. No speedup or possibly a very slight speedup was observed by this.
- Performing as little as possible work in general. Some significant speedup were observed by doing this.

Parallel optimization used:

- Performing as much work as possible in parallel. Significant speedup was observed by doing this.
- Minimizing the number of parallel sections, i.e. having one or only a few `#pragma omp parallel` with several loops instead of several separate `#pragma omp parallel`. Significant speedup was observed by doing this, probably due to decreased overhead cost.
- During the thread grouping, two independent for-loops are used. Here, OpenMP keyword `nowait` was added to improve synchronization. Some speedup was observed by this. `nowait` was also used in other parts where applicable but no or possibly only a very slight speedup was observed by this.
- OpenMP keyword `schedule(auto)` provided a significant speedup and was used in every for-loop. `schedule(dynamic)` gave the same speedup, indicating that `schedule(auto)` chooses to use `type=dynamic`. `schedule(static)` resulted in significantly slower execution times. `schedule(dynamic)` probably improved the load balance significantly as the parallel task are not equally expensive, hence the speedup. Another advantage of using `schedule(auto)` is that we do not need to specify `size` explicitly which is required in order for some loops to work, resulting in code that is easier to read.

Different compiler flags were tried but none resulted in any large performance improvements:

- Using `-O2`, `-O3`, or `-Ofast` entailed a very slight speedup compared to `-O1`. There were no observable difference between `-O2`, `-O3`, and `-Ofast`.
- `-march=native` with `-O3` or `-Ofast` entailed a very slight speedup.

3.1.2 Optimizations of the algorithm

The parallel algorithm can be implemented so that data dependencies and synchronization are minimized. With regards to the data dependencies, the algorithm only needs access to one other thread in each thread when well implemented. Synchronization can then be minimized through utilizing the fact that each thread creates one sorted sub-array from two other sub-arrays, and each thread access unique sub-arrays. As there is no data overlap between threads, each recursive iteration can be executed in parallel. However, this comes at a cost of allocating new arrays and freeing old arrays every iteration.

Besides implementing the algorithm well in general, significant performance improvements can be acquired by a good pivot. A good pivot is an element that is close to the median of the array or sub-array to sort. Choosing such a pivot results in optimal partitions and for the parallel case, an even load balance. The pivot selection during the quicksorting in each thread is done by `qsort`, but the pivot during the thread dividing still needs to be chosen somehow. A simple strategy is to select the median element in an arbitrary thread, i.e. the first thread, which yields good performance if the array is randomly distributed. However, if the array is approximately sorted already, this can result in bad load balance due to the pivot deviating significantly from the median. A solution would be to choose the median of the medians from all threads. However, this extra work to choose the pivot results in slightly worse performance than the arbitrary median element strategy if the array is randomly distributed. For an approximately sorted array this solution is slightly faster. In the implementation in this report, the simple strategy was chosen because it performs better for arbitrary arrays. Nearly sorted arrays can be considered extreme cases.

4 Experiments

The timings were run on:

- Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz processor (with 2 cores) in macOS Mojave version 10.14.3 (18D109). The Apple LLVM version 10.0.0 (clang-1000.11.45.5) compiler was used, with parallelization using OpenMP. The timings performed on this processor will be referred to as the timings on the 2 cores computer.
- AMD Opteron(tm) Processor 6282 SE @ 2600.102 MHz (with 16 cores) in x86_64 GNU/Linux on the Uppsala University server `vitsippa.it.uu.se`. The GCC 4.4.7 20120313 (Red Hat 4.4.7-23) compiler was used, with parallelization using OpenMP. The timings performed on this processor will be referred to as the timings on the 16 cores computer.

Timings were performed using the `gettimeofday()` function defined in `<sys/time.h>` before and after the parallel quicksort. After every sort and timing, the array was checked to be correctly sorted. Timings were performed on random integers between 0 and 999 generated by `rand() % 1000` where `rand()` is defined in `<stdlib.h>`.

4.1 Timings

4.1.1 Before any optimizations

Sorting 100 million integers took 4.30 seconds with 4 threads on the 2 cores computer.

4.1.2 After optimizations

Sorting 100 million integers took 2.66 seconds with 4 threads on the 2 cores computer. This is an obvious improvement (speedup of 1.6) to the 4.30 seconds it took before the optimizations with 4 threads on the 2 cores computer.

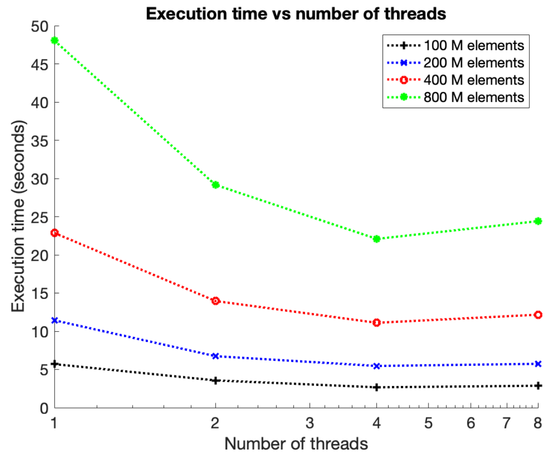
4.2 Execution time and speedup

4.2.1 Timings on the 2 cores computer

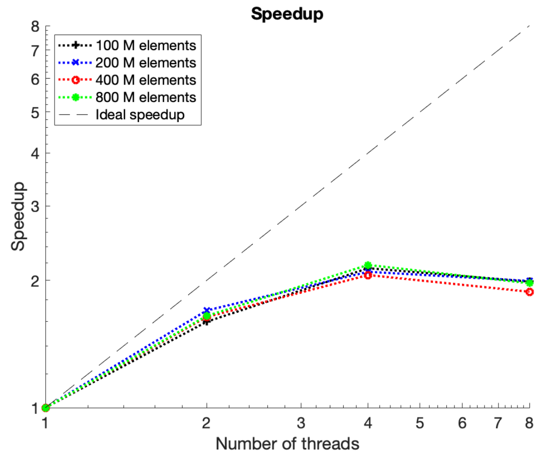
Execution timings and speedup for different number of threads and different number of elements to sort on the 2 cores computer are presented in figure 1. In figure 1, it can be observed that the execution time was the lowest and the speedup was the highest using 4 threads regardless of how many elements were sorted.

4.2.2 Timings on the 16 cores computer

Execution timings and speedup for different number of threads and different number of elements to sort on the 16 cores computer are presented in figure 2. In figure 2, it can be observed that the execution time was the lowest and the speedup was the highest using 32 or 64 threads. When using a high number of threads (more than 16 threads), the speedup was greater when sorting a larger number of elements than when sorting a smaller number of elements.

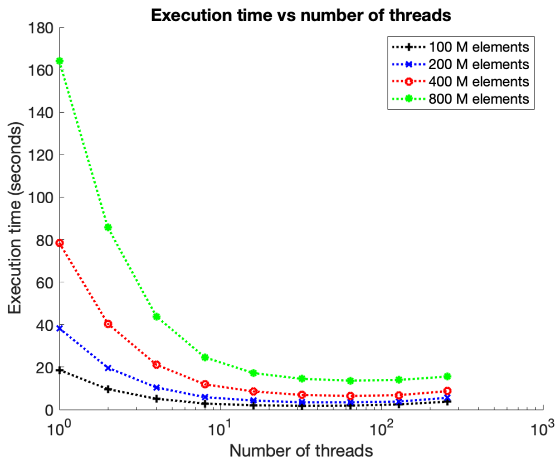


(a) Execution time

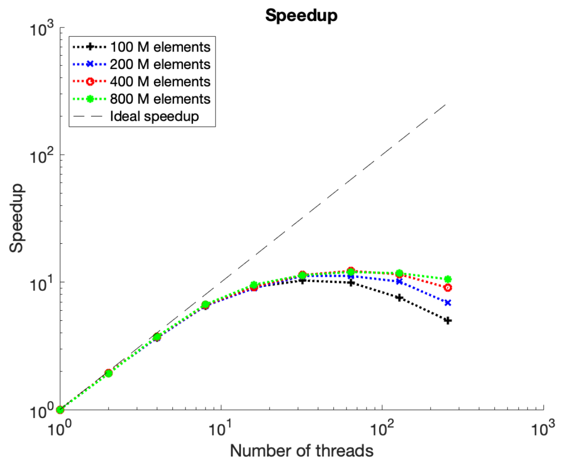


(b) Speedup

Figure 1: Execution time and speedup of the parallel quicksort implementation for different number of threads and different number of elements to sort on the 2 cores computer. The execution time is lowest and the speedup is highest (just below 2) using 4 threads regardless of how many elements were sorted.



(a) Execution time



(b) Speedup

Figure 2: Execution time and speedup of the parallel quicksort implementation for different number of threads and different number of elements to sort on the 16 cores computer. The execution time is lowest and the speedup is highest (approximately 10) using 32 or 64 threads regardless of how many elements were sorted.

5 Conclusions

The execution time is lowest and the speedup is highest using 4 threads and 32 or 64 threads on the 2 cores and 16 cores computer respectively, regardless of how many elements were sorted. This is expected as the threads used are the same as or greater than the number of cores. For the 2 cores computer, although a speedup of 2 is significant, it is far from the ideal speedup of 4 using 4 threads. The highest speedup of approximately 12 for the 16 cores computer is better but still far from the ideal speedup of 32 using 32 threads, or 64 using 64 threads. Deviations from the ideal speedups aside, the parallel speedup is definitely significant and useful in practice.

An interesting thing to note is that the speedup when using a high number of threads (more than 16 threads), the speedup was greater when sorting a larger number of elements than when sorting a smaller number of elements. For example, when using 256 cores, the speedup was around 4 when sorting 100 M elements but the speedup was around 10 when sorting 800 M elements. This might be because of a better load balance. The high number of cores have an easier task of load balancing 800 M elements than 100 M elements.

The speedup was unsurprisingly greater on the 16 cores computer than on the 2 cores computer. On the 2 cores computer, the observed speedup differed from the ideal speedup a lot more than on the 16 cores computer. On the 16 cores computer, the observed speedup was reasonably close to the ideal speedup when using 2, 4, and 8 threads. The performance of the implemented parallel quicksort algorithm was overall better on the 16 cores computer than on the 2 cores computer, indicating that the algorithm makes good use of parallelism.

In this implementation, the array is copied during each recursion. A possible improvement would be to skip this presumably costly step and make use of the original, already allocated array instead by carefully controlling relevant indices for the sub-arrays. The main reason this possible improvement is not implemented in this report is that this would require reading and writing from several threads, sacrificing synchronization improvements. It was also deemed a more convoluted implementation.

Another improvement of the implemented parallel quicksort algorithm would be to make it work for any data type as `qsort` does and not just integers. This would not result in any performance improvements with regards to the parallelism but could be useful if the algorithm is to be used in practice and not just evaluated as in this report.

References

[1] Quicksort Wikipedia article. URL (retrieved 2019-03-14): `https://en.wikipedia.org/wiki/Quicksort`

[2] Lectures about parallel sorting, Jarmo Rantakokko, High Performance Computing course, March 2019. Department of Information Technology, Uppsala University, Sweden.